Appendix A

Travel.cpp

Attorney Docket 3296.1

Inventor: Earl A. Hubbell


```
#include <cstring.h>
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define TAB '\t'
#define ARRAYHASH 256


char base_from_num(long num)
{
        switch (num%4)
        {
                case 0: return('A');
                case 1: return('C');
                case 2: return('G');
                case 3: return('T');
                default: return ('A');
        }
}

long transform(string &Test)
{
        Test.to_upper();
        for (long i=0; i<Test.length(); i++)
   {
        if (Test[i]=='A')
        Test[i]='T';
                else
              if (Test[i]=='C')
                Test[i] = 'G';
         else
                      if (Test[i]=='G')
                        Test[i]='C';
            else
                              if (Test[i]=='T')
                                              Test[i]='A';
        }
   return(TRUE);
}


class ProbeSynth{
```

```
public:
        string Probe;
        string Original;
        string Name;
        string Rename;
        long Location;
        long Unit;
        long Atom;
        // done with fancy
        char *Synth;
        long SynthLength;
        // shift by 4,8,12
        long SynthModifier;
   long MutVal; // mutation position
        ProbeSynth();
   ~ProbeSynth();
   Destroy();
        Allocate(long);
   Synthesize();
   SynthesizeFluff(long);
   SynthesizeFastFluff(long);
   SynthesizeMutant(long);
   SynthesizeMisMatch(long);
   CompSynth();
   char GetSynth(long);
        SetSynth(long, char);
   ZeroCost();
   Distance(ProbeSynth *);
        Output(ofstream &);
        List(ofstream &);
};

ProbeSynth::SynthesizeMutant(long MutateValue)
{
        long synthflag, done, fwdflag;
        long probecount, cyclecount, qloop;
   long start, finish;
        long resynth;

   start = 0;
        finish = 0;
   Allocate(4*Probe.length()+4);
        probecount = 0;
        this->MutVal = MutateValue;
        cyclecount = start;
        done = FALSE;
        synthflag = FALSE;
        fwdflag = TRUE;

        while (!done)
        {
                if (probecount==MutateValue)
                {
                        if (probecount<Probe.length()-1)
                        {
                                resynth = cyclecount;
                                cyclecount=resynth+5; /*leave room for mutant*/
                                /*synthesize mutant*/
                                synthflag = FALSE;
                                for (qloop=resynth; qloop<cyclecount-1; qloop++)
                                {
                                        if(base_from_num(qloop)==Probe[probecount])
                                        {
                        // let us know that this is a mutant (and hence different)
                   // >if< we were doing ACGT, filling in these with all 4
                   // would let us match probes well, since geographic position
                   // would matter.  But we're only doing single mismatches,
                   // for GE, so that doesn't work.
                                                SetSynth(qloop, '*');
                                                synthflag = TRUE;
                                        }
```

```
                                        else
                        SetSynth(qloop, '*');
                                }
                                if (!synthflag)
                                        done = TRUE;
                                probecount++;
                cyclecount--;
                                /*cancel out cyclecount++ later*/
                        }
                        else
                        {
                                if (base_from_num(cyclecount)== Probe[probecount])
                                {
                                        SetSynth(cyclecount, base_from_num(cyclecount));
                                        synthflag = TRUE;
                                        probecount++;
                                }
                        }
                        if (probecount>Probe.length()-1)
                                done = TRUE;

                        if (finish>0 && synthflag)
                        {
                                fwdflag = FALSE;
                                probecount = Probe.length()-1;
                                cyclecount = finish;
                        }
                        else
                        {
                   cyclecount++;
                        }
                   }
                   else
                        if (base_from_num(cyclecount)==Probe[probecount])
                        {
                                if (fwdflag)
                                {
                                        SetSynth(cyclecount, base_from_num(cyclecount));
                                        probecount++;
                                        cyclecount++;
                                        if (probecount>=Probe.length())
                                                done = TRUE;
                                }
                                else
                                {
                                        SetSynth(cyclecount, base_from_num(cyclecount));
                                        probecount--;
                   cyclecount--;
                                        if (probecount<=MutateValue)
                                                done = TRUE;
                                }
                        }
                        else
                        {
                                if (fwdflag)
                   cyclecount++;
                                else
                   cyclecount--;
                        }
        }
        this->SynthLength = cyclecount; // shorten 'search' space
        return(TRUE);
}

ProbeSynth::SynthesizeMisMatch(long MutateValue)
{
        long synthflag, done, fwdflag;
        long probecount, cyclecount, qloop;
        long start, finish;
        long resynth;
```

```
start = 0;
finish = 0;
     Allocate(4*Probe.length()+4);
     probecount = 0;
     cyclecount = start;
this->MutVal = MutateValue;
     done = FALSE;
     synthflag = FALSE;
     fwdflag = TRUE;

     while (!done)
     {
            if (probecount==MutateValue)
            {
                   if (probecount<Probe.length()-1)
                   {
                          resynth = cyclecount;
                          cyclecount=resynth+5; /*leave room for mutant*/
                          /*synthesize mutant*/
                          synthflag = FALSE;
                          for (qloop=resynth; qloop<cyclecount-1; qloop++)
          {
                                 if(base_from_num(qloop)==Probe[probecount])
                                 {
                                        // let us know that this is a mutant (and hence
different)
                                        SetSynth(qloop, '#');
                                        synthflag = TRUE;
                                 }
          }
                          if (!synthflag)
                                 done = TRUE;
                          probecount++;
          cyclecount--;
                          /*cancel out cyclecount++ later*/
                   }
                   else
                   {
                          if (base_from_num(cyclecount)== Probe[probecount])
                          {
                                 SetSynth(cyclecount, base_from_num(cyclecount));
                                 synthflag = TRUE;
                                 probecount++;
                          }
                   }
                   if (probecount>Probe.length()-1)
                          done = TRUE;

                   if (finish>0 && synthflag)
                   {
                          fwdflag = FALSE;
                          probecount = Probe.length()-1;
                          cyclecount = finish;
                   }
                   else
                   {
                          cyclecount++;
                   }
            }
            else
                   if (base_from_num(cyclecount)==Probe[probecount])
                   {
                          if (fwdflag)
                          {
                                 SetSynth(cyclecount, base_from_num(cyclecount));
                                 probecount++;
          cyclecount++;
                                 if (probecount>=Probe.length())
                                        done = TRUE;
                          }
                          else
```

```
                                {
                                        SetSynth(cyclecount, base_from_num(cyclecount));
                                        probecount--;
                        cyclecount--;
                                        if (probecount<=MutateValue)
                                                done = TRUE;
                                }
                        }
                        else
                        {
                                if (fwdflag)
                        cyclecount++;
                                else
                                        cyclecount--;
                        }
                }
                this->SynthLength = cyclecount; // shorten 'search' space
                return(TRUE);
        }

        char
        ProbeSynth::GetSynth(long Which)
        {
                if (Which < SynthLength && Which>-1)
                        return(Synth[Which]);
                else
                        return('.');
        }

        ProbeSynth::SetSynth(long Which, char What)
        {
                if (Which>-1 && Which < SynthLength)
                Synth[Which] = What;
           return(TRUE);
        }

        ProbeSynth::ZeroCost()
        {
                return(Probe.length());
        }

        ProbeSynth::Distance(ProbeSynth *Destination)
        {
                static char testchar;
                static long count;
                static long minlen;
                static long i;

                count = 0;
                minlen = min(Destination->SynthLength, this->SynthLength);

                for (i=0; i<minlen; i++)
                {
                        testchar = Destination->Synth[i];  // can guarantee i>0
                        if (testchar!='.')
                                if (this->Synth[i]!=testchar)
                                {
                                        if (testchar == '#')
                                                count+=2;  // heavily penalize mutants
                                        else
                                                count++;
                                }
                }
                for (; i<Destination->SynthLength; i++)
                {
                        testchar = Destination->Synth[i];  // can guarantee i>0
                        if (testchar!='.')
                        {
                                if (testchar == '#')
                                        count+=2;  // heavily penalize mutants
                                else
```

```
                                count++;
                        }
                }
                return(count);
        }

        ProbeSynth::Synthesize()
        {
                long Size = CompSynth();
                Allocate(Size);
                long length, probeloop, synthloop;
                char testchar;

                length = Probe.length();
                probeloop = 0;
                synthloop = 0;
                while (probeloop<length)
                {
                        testchar = base_from_num(synthloop);
                        if (testchar==Probe[probeloop])
                        {
                                Synth[synthloop] = Probe[probeloop];
                                probeloop++;
                        }
                        else
                                Synth[synthloop] = '.';
                        synthloop++;
                }
                return(TRUE);
        }

        ProbeSynth::SynthesizeFluff(long MutateValue)
        {
                long length, probeloop, synthloop, testloop;
                char testchar;

                length = Probe.length();
                long Size = 4*length+4;
                Allocate(Size);
           this->MutVal = MutateValue;
                probeloop = 0;
                synthloop = 0;
           for (probeloop=0; probeloop<length; probeloop++)
                {
                        synthloop= probeloop*4;
              testchar = Probe[probeloop];
                for (testloop = 0; testloop<4; testloop++, synthloop++)
                        {
                                if (base_from_num(synthloop)==testchar)
                  {
                        if (probeloop!=MutateValue)
                                                Synth[synthloop] = testchar;
                                else
                        Synth[synthloop] = '#';
                  }
                                else
                        Synth[synthloop] = '.';
              }
                }
                return(TRUE);
        }

        ProbeSynth::SynthesizeFastFluff(long MutateValue)
        {
                long length, probeloop, synthloop, testloop;
                char testchar;

                length = Probe.length();
                long Size = length+1;
                Allocate(Size);
           this->MutVal = MutateValue;
```

```
        probeloop = 0;
        synthloop = 0;
    for (probeloop=0, synthloop = 0; probeloop<length; probeloop++, synthloop++)
        {
                Synth[synthloop] = Probe[probeloop];
       if (probeloop==MutateValue)
       {
        synthloop++; // account for mutant being twice as bad
         Synth[synthloop]=Probe[probeloop];
                }
        }
        return(TRUE);
}

ProbeSynth::CompSynth()
{
        long length, probeloop, synthloop;
        char testchar;

        length = Probe.length();
        probeloop = 0;
        synthloop = 0;
        while (probeloop<length)
        {
                testchar = base_from_num(synthloop);
                if (testchar==Probe[probeloop])
                {
                        probeloop++;
                }
                synthloop++;
        }

        return(synthloop);
}


ProbeSynth::ProbeSynth()
{
        Probe = "";
    Name = "None";
        Location = 0;
    Synth = NULL;
        SynthLength = 0;
        SynthModifier = 0;
}

ProbeSynth::Allocate(long Size)
{
        Destroy();
    Synth = new char [Size];
    for (long i=0; i<Size; i++)
                Synth[i] = '.';
    SynthLength = Size;
}

ProbeSynth::Destroy()
{
        if (Synth!=NULL)
    {
        delete[] Synth;
       Synth = NULL;
       SynthLength = 0;
    }
}

ProbeSynth::~ProbeSynth()
{
        Destroy();
}

ProbeSynth::Output(ofstream &OutStream)
```

```
{
        OutStream << this->Location << TAB;
        OutStream << this->Original << TAB;
        OutStream << this->Name << TAB;
        OutStream << this->Rename << TAB;
        OutStream << this->Unit << TAB;
        OutStream << this->Atom << endl;
}

ProbeSynth::List(ofstream &OutStream)
{
        OutStream << this->Original << endl;
}

class ProbeNode{
public:
        ProbeSynth *DataPointer;
   ProbeNode *Previous;
   ProbeNode *Next;
        ProbeNode **PClosest;
        long plength;
        long marker;
        long NextCost;
        PointToData(ProbeSynth *);
        ProbeNode();
   ~ProbeNode();
   Destroy();
   DestroyPClosest();
        DestroyData();
   AllocatePClosest(long);
        LinkPrevious(ProbeNode *);
   LinkNext(ProbeNode *);
   InsertNext(ProbeNode *);
   Initialize();
        ZeroCost();
   Copy(ProbeNode *);
   Distance(ProbeNode *);
};

ProbeNode::Copy(ProbeNode *Original)
{
        if (Original!=NULL)
                DataPointer = Original->DataPointer;
        else
                return(FALSE);
   // note - do not copy Pclosest, plength - don't apply to copies
   // do not copy previous & next, because they won't correspond.
}


ProbeNode::PointToData(ProbeSynth *Data)
{
        DataPointer = Data;
}

ProbeNode::ZeroCost()
{
        return(DataPointer->ZeroCost());
}

ProbeNode::Distance(ProbeNode *Destination)
{
        return(DataPointer->Distance(Destination->DataPointer));
}

ProbeNode::ProbeNode()
{
        DataPointer = NULL;
   Previous = NULL;
   Next = NULL;
        PClosest = NULL;
```

```
   marker = 0;
        plength =0;
   NextCost = 0;
}

ProbeNode::Destroy()
{
        DataPointer = NULL;
   Previous = NULL;
   Next = NULL;
}

ProbeNode::DestroyPClosest()
{
        if (PClosest !=NULL)
        {
        delete[] PClosest;
      PClosest = NULL;
      plength = 0;
        }
}

ProbeNode::DestroyData()
{
        // dangerous - must be last called
   if (DataPointer!=NULL)
   {
                delete[] DataPointer;
      DataPointer = NULL;
        }
}

ProbeNode::AllocatePClosest(long Size)
{
        DestroyPClosest();
        PClosest = new ProbeNode * [Size];
   for (long i=0; i<Size; i++)
        PClosest[i] = NULL;
   return(TRUE);
}

ProbeNode::~ProbeNode()
{
        Destroy();
}

ProbeNode::LinkPrevious(ProbeNode *Link)
{
        Previous = Link;
}

ProbeNode::LinkNext(ProbeNode *Link)
{
        Next = Link;
}

ProbeNode::InsertNext(ProbeNode *NewNode)
{
        ProbeNode *Link;

   Link = Next;
        Next = NewNode;
   NewNode->Previous = Link->Previous;
   Next->Previous = NewNode;
   NewNode->Next = Link;
}

ProbeNode::Initialize()
{
        Previous = this;
   Next = this;
```

```
}

class TourClass{
public:
        ProbeNode *DataList;
   long Cost;
        TourClass();
        InitializeTour(ProbeNode *);
        DeleteCurrent();
        UnlinkCurrent();
        DestroyList();
        InsertAfterCurrent(ProbeNode *);
        long TestBasicInsertion(ProbeNode *);
        Rotate();
        Duplicate(TourClass &);
        InsertLeastCost(ProbeNode *);
        InsertLeastCostFromPool(TourClass &);
        InsertLeastCostByMarker(TourClass &);
        InsertLeastCostWithModifier(ProbeNode *, long);
        QuickInsertLeastCost(ProbeNode *, long);
        ~TourClass();
        Output(ofstream &);
        List(ofstream &);
};

TourClass::TourClass()
{
        DataList = NULL;
        Cost = 0;
}

TourClass::Output(ofstream &OutStream)
{
        ProbeNode *Start = this->DataList;

        this->DataList->DataPointer->Output(OutStream);
        Rotate();
        while (Start != this->DataList)
        {
                this->DataList->DataPointer->Output(OutStream);
                Rotate();
        }
}

TourClass::List(ofstream &OutStream)
{
        ProbeNode *Start = this->DataList;

        this->DataList->DataPointer->List(OutStream);
        Rotate();
        while (Start != this->DataList)
        {
                this->DataList->DataPointer->List(OutStream);
                Rotate();
        }
}

TourClass::Duplicate(TourClass &TestTour)
{
        ProbeNode *Duplicate;
        ProbeNode *Start;

   this->DestroyList();
        Start = TestTour.DataList;
   Duplicate = new ProbeNode;
   Duplicate->Copy(TestTour.DataList);
        InitializeTour(Duplicate);
   TestTour.Rotate();

   while (Start!=TestTour.DataList)
```

```
        {
        Duplicate = new ProbeNode;
      Duplicate->Copy(TestTour.DataList);
      InsertAfterCurrent(Duplicate);
              TestTour.Rotate();
      Rotate();
        }
   return(TRUE);
}

TourClass::InsertAfterCurrent(ProbeNode *NewNode)
{
        ProbeNode *Link;

        if (DataList==NULL)
        {
                InitializeTour(NewNode);
                return(TRUE);
        }
        Link = DataList->Next;
        DataList->Next = NewNode;
        Link->Previous = NewNode;
        NewNode->Next = Link;
        NewNode->Previous = DataList;
        Cost = Cost - DataList->NextCost;
        DataList->NextCost = DataList->Distance(NewNode);
        Cost = Cost + DataList->NextCost;
        NewNode->NextCost = NewNode->Distance(Link);
        Cost = Cost + NewNode->NextCost;
}

long
TourClass::TestBasicInsertion(ProbeNode *NewNode)
{
        long TestCost;

        TestCost = Cost - DataList->NextCost;
        TestCost += DataList->Distance(NewNode);
   TestCost += NewNode->Distance(DataList->Next);
   return(TestCost);
}

TourClass::Rotate()
{
        DataList = DataList->Next;
}

TourClass::InsertLeastCost(ProbeNode *NewNode)
{
        ProbeNode *Start;
        ProbeNode *BestPlace;
   long TestCost, BestCost;

   Start = DataList;
        BestPlace = DataList;
   TestCost = TestBasicInsertion(NewNode);
        BestCost = TestCost;
        Rotate();
   while (DataList!=Start)
        {
                TestCost = TestBasicInsertion(NewNode);
      if (TestCost<BestCost)
      {
        BestCost = TestCost;
          BestPlace = DataList;
      }
      Rotate();
   }
        DataList = BestPlace;
   InsertAfterCurrent(NewNode);
}
```

```
TourClass::InsertLeastCostByMarker(TourClass &Source)
{
        // searches Source for undone ones,
    // adds the closest one to the new tour
        ProbeNode *Start;
    ProbeNode *BestPlace, *BestAdd;
    ProbeNode *NewNode, *Done;
        long TestCost, BestCost;

        Start = DataList;
        BestPlace = DataList;
    NewNode = Source.DataList;
        Done = Source.DataList;
    TestCost = TestBasicInsertion(NewNode);
    BestCost = TestCost;
    Rotate();
        Source.Rotate();
    long totalmarker = 0;
        while(Source.DataList!=Done);
    {
                NewNode = Source.DataList;
           while (DataList!=Start && !NewNode->marker)
                {
                        TestCost = TestBasicInsertion(NewNode);
                        if (TestCost<BestCost)
                {
                                BestAdd = NewNode;
                  BestCost = TestCost;
                    BestPlace = DataList;
                        }
                        Rotate();
                        totalmarker = 1;
           }
       Source.Rotate();
    }
    NewNode = new ProbeNode;
    NewNode->Copy(BestAdd);
    BestAdd->marker = 1;
        DataList = BestPlace;
    InsertAfterCurrent(NewNode);
        return(totalmarker);
}

TourClass::InsertLeastCostFromPool(TourClass &Source)
{
        // searches Source for undone ones,
    // adds the closest one to the new tour
        ProbeNode *Start;
    ProbeNode *BestPlace, *BestAdd;
    ProbeNode *NewNode, *Done;
    long TestCost, BestCost;

        BestAdd = NULL;
        BestPlace = NULL;
        NewNode = Source.DataList;
        Done = Source.DataList;
        TestCost = 1000;
        BestCost = TestCost;
        Source.Rotate();
        long totalmarker = 0;
        while(Source.DataList!=Done)
        {
                NewNode = Source.DataList;
                //cout << NewNode->DataPointer->Probe << TAB << NewNode->marker << endl;
                if (!NewNode->marker)
                {
                        TestCost = TestBasicInsertion(NewNode);
                        if (TestCost<BestCost)
                        {
                                BestAdd = NewNode;
```

```
                        BestCost = TestCost;
                    }
                    totalmarker = 1;
                }
                Source.Rotate();
            }
            if (totalmarker && BestAdd)
            {
                NewNode = new ProbeNode;
                NewNode->Copy(BestAdd);
                BestAdd->marker = 1;
                InsertAfterCurrent(NewNode);
                cout << NewNode->DataPointer->Probe << endl;
                Rotate(); // go to NewNode as the favorite
            }
            return(totalmarker);
}

TourClass::InsertLeastCostWithModifier(ProbeNode *NewNode, long ModMax)
{
            ProbeNode *Start;
            ProbeNode *BestPlace;
            long BestModifier;
            long TestCost, BestCost;

            Start = DataList;
            BestPlace = DataList;
            TestCost = TestBasicInsertion(NewNode);
            BestCost = TestCost;
            BestModifier = 0;

            for (long ModLoop = 0; ModLoop<ModMax; ModLoop++)
            {
                NewNode->DataPointer->SynthModifier = ModLoop;
                Start = DataList;
                Rotate();
                while (DataList!=Start)
                {
                    TestCost = TestBasicInsertion(NewNode);
                    if (TestCost<BestCost)
                    {
                        BestCost = TestCost;
                        BestPlace = DataList;
                        BestModifier = ModLoop;
                    }
                    Rotate();
                }
            }
            DataList = BestPlace;
            NewNode->DataPointer->SynthModifier = BestModifier;
            InsertAfterCurrent(NewNode);
}

TourClass::QuickInsertLeastCost(ProbeNode *NewNode, long SearchLevel)
{
            ProbeNode *Start;
            ProbeNode *BestPlace;
            long TestCost, BestCost;

            Start = DataList;
            BestPlace = DataList;
            TestCost = TestBasicInsertion(NewNode);
            BestCost = TestCost;

            Start = DataList;
            Rotate();
            long counter = 0;
            NewNode->DataPointer->SynthModifier = 0;
            while (DataList!=Start && counter<SearchLevel)
            {
                TestCost = TestBasicInsertion(NewNode);
```

```
                if (TestCost<BestCost)
                {
                        BestCost = TestCost;
                        BestPlace = DataList;
                }
                Rotate();
                counter++;
        }
        DataList = BestPlace;
        InsertAfterCurrent(NewNode);
}

TourClass::DeleteCurrent()
{
        ProbeNode *Link;
        ProbeNode *Del;

        Link = DataList->Next;
   Del = DataList;
        Cost = Cost - DataList->NextCost;
        Cost = Cost - DataList->Previous->NextCost;
   if (Link==DataList)
   {
        DataList = NULL;
      delete Del;
                Cost = 0;
      return(TRUE);
        }
   Link->Previous = DataList->Previous;
   Link->Previous->Next = DataList->Next;
   DataList = Link;
        delete Del;
   Link->Previous->NextCost = Link->Previous->Distance(Link);
   Cost = Cost + Link->Previous->NextCost;
   return(TRUE);
}

TourClass::UnlinkCurrent()
{
        ProbeNode *Link;
   ProbeNode *Del;

        Link = DataList->Next;
   Del = DataList;
   Cost = Cost - DataList->NextCost;
        Cost = Cost - DataList->Previous->NextCost;
   if (Link==DataList)
   {
        DataList = NULL;
                Cost = 0;
      return(TRUE);
   }
   Link->Previous = DataList->Previous;
        Link->Previous->Next = DataList->Next;
   DataList = Link;
        Link->Previous->NextCost = Link->Previous->Distance(Link);
   Cost = Cost + Link->Previous->NextCost;
        return(TRUE);
}

TourClass::InitializeTour(ProbeNode *Test)
{
        DestroyList();
        DataList = Test;
        Test->Initialize();
        Cost = Test->ZeroCost();
        Test->NextCost = 0;
        return(TRUE);
}

TourClass::DestroyList()
```

```
{
        while (DataList!=NULL)
                DeleteCurrent();
}


TourClass::~TourClass()
{
        DestroyList();
}

class TSPClass{
public:
        TourClass DataSet;
        TourClass BestTour;
        TourClass CurrentTour;
        TSPClass();
        ~TSPClass();
        LoadData(string);
        LoadMutantData(string, long);
        LoadMisMatchData(string, long);
        LoadMisMatchFluffData(string, long);
        LoadMisMatchFastFluffData(string, long);
        LoadExpressionData(string, long);
        LoadSingleExpressionData(string);
        LoadExpressionDataByUnit(string, long, long);
        LoadExpressionFluffData(string, long);
        LoadChipData(string);
        GenerateTourByInsertion();
        GenerateTourByInsertionAndDeletion();
        GenerateTourByClosestInsertion();
        GenerateTourByClosestPool();
        GenerateTourByInsertionWithModifier(long);
        GenerateQuickTourByInsertion(long);
        ImproveTourByReplacement(long);
        OutputTour(string);
        AppendTour(string);
        ListTour(string);
        DestroyData();
//   Geni(int);
};

TSPClass::TSPClass()
{
}

TSPClass::DestroyData()
{
        ProbeNode *DataStart;

        DataStart = DataSet.DataList;
        DataSet.DataList->DestroyData();
        DataSet.Rotate();

   while(DataStart!=DataSet.DataList)
   {
          DataSet.DataList->DestroyData();
              DataSet.Rotate();
   }
        return(TRUE);
}

TSPClass::~TSPClass()
{
        //DestroyData();
}

TSPClass::GenerateTourByInsertion()
{
        ProbeNode *DataStart;
        ProbeNode *TempData;
```

```
        DataStart = DataSet.DataList;
        TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
        BestTour.InitializeTour(TempData);
        DataSet.Rotate();

        long counter = 0;
        while(DataStart!=DataSet.DataList)
        {
                TempData = new ProbeNode;
                TempData->Copy(DataSet.DataList);
                //cout << DataSet.DataList->DataPointer->Probe << TAB << counter << TAB;
                BestTour.InsertLeastCost(TempData);
                //cout << BestTour.Cost << endl;
                DataSet.Rotate();
                if (counter%100==0)
                        cout << counter << endl;
                counter++;
        }
        return(TRUE);
}


TSPClass::GenerateTourByInsertionAndDeletion()
{
        ProbeNode *DataStart;
        ProbeNode *TempData;


        DataStart = DataSet.DataList;
        if (DataSet.DataList->marker>0)
                DataSet.Rotate();
        while (DataSet.DataList->marker>0 && DataSet.DataList!=DataStart)
        {
                DataSet.Rotate();
        }
        if (DataSet.DataList->marker>0)
                return(FALSE);
        DataStart = DataSet.DataList;
        DataStart->marker = 1;

        TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
        BestTour.InitializeTour(TempData);

        long Unit = TempData->DataPointer->Unit;

        DataSet.Rotate();

        long counter = 0;
        while(DataStart!=DataSet.DataList)
        {
                if (DataSet.DataList->DataPointer->Unit==Unit && DataSet.DataList->marker<1)
                {
                        TempData = new ProbeNode;
                        TempData->Copy(DataSet.DataList);
                        cout << DataSet.DataList->DataPointer->Name << TAB << DataSet.DataList-
>DataPointer->Unit << TAB << counter << endl;
                        BestTour.InsertLeastCost(TempData);
                        DataSet.DataList->marker = 1;
                //cout << BestTour.Cost << endl;
                }
                DataSet.Rotate();
                counter++;
                if (counter%1000==0)
                        cout << counter << endl;
        }
        return(TRUE);
}
```

```
TSPClass::GenerateTourByClosestInsertion()
{
        ProbeNode *DataStart;
   ProbeNode *TempData;

        DataStart = DataSet.DataList;
   TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
   DataSet.DataList->marker = 1; // set on this course
        BestTour.InitializeTour(TempData);

        long notdone = 1;
        long counter = 0;
   while (notdone)
        {
                notdone = BestTour.InsertLeastCostByMarker(DataSet);
      if (counter%100==0)
             cout << counter << endl;
              counter++;
   }
        return(TRUE);
}

TSPClass::GenerateTourByClosestPool()
{
        ProbeNode *DataStart;
        ProbeNode *TempData;

        DataStart = DataSet.DataList;
        TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
        DataSet.DataList->marker = 1; // set on this course
        BestTour.InitializeTour(TempData);

        long notdone = 1;
        long counter = 0;
        while (notdone)
        {
                notdone = BestTour.InsertLeastCostFromPool(DataSet);
                counter++;
        }
        return(TRUE);
}

TSPClass::GenerateTourByInsertionWithModifier(long ModMax)
{
        ProbeNode *DataStart;
        ProbeNode *TempData;

        DataStart = DataSet.DataList;
        TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
        BestTour.InitializeTour(TempData);
        DataSet.Rotate();

        long counter = 0;
        while(DataStart!=DataSet.DataList)
        {
                TempData = new ProbeNode;
                TempData->Copy(DataSet.DataList);
                BestTour.InsertLeastCostWithModifier(TempData, ModMax);
                if (counter%100==0)
                {
                        cout << DataSet.DataList->DataPointer->Probe << TAB << counter << TAB;
                        cout << BestTour.Cost << endl;
                }
                DataSet.Rotate();
                counter++;
        }
        return(TRUE);
}
```

```
TSPClass::GenerateQuickTourByInsertion(long SearchLevel)
{
        ProbeNode *DataStart;
        ProbeNode *TempData;

        DataStart = DataSet.DataList;
        TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
        BestTour.InitializeTour(TempData);
        DataSet.Rotate();

        long counter = 0;
        while(DataStart!=DataSet.DataList)
        {
                TempData = new ProbeNode;
                TempData->Copy(DataSet.DataList);
                BestTour.QuickInsertLeastCost(TempData, SearchLevel);
                if (counter%100==0)
                {
                        cout << DataSet.DataList->DataPointer->Probe << TAB << counter << TAB;
                        cout << BestTour.Cost << endl;
                }
                DataSet.Rotate();
                counter++;
        }
        return(TRUE);
}

TSPClass::ImproveTourByReplacement(long ReplaceSize)
{
        ProbeNode *DataStart;
        ProbeNode *TempData;


   long counter = 0;
        while(counter<ReplaceSize)
        {
                TempData = BestTour.DataList;
                if (TempData->marker<1)
      {
                        TempData->marker = 1;
                        BestTour.UnlinkCurrent();
              cout << TempData->DataPointer->Probe << TAB << counter << TAB;
              BestTour.InsertLeastCost(TempData);
           cout << BestTour.Cost << endl;
                }
      else
                        BestTour.Rotate();
      counter++;
        }
        return(TRUE);
}

TSPClass::LoadData(string FileName)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
   string TestString;
   ProbeNode *NewNode;
        float test; // soak up number
        string junkstring;
```

```
        while (!ExpStream.eof())
        {
                ExpStream >> TestString >> junkstring >> test;
                if (TestString.length()>1)
       {
                        NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Original = TestString;
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->Synthesize();
                        this->DataSet.InsertAfterCurrent(NewNode);
                        //cout << TestString << endl;
                }
        }
        ExpStream.close();
        cout << DataSet.Cost << endl;
}

TSPClass::LoadMutantData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
        string TestString;
        ProbeNode *NewNode;
        float test; // soak up number

        while (!ExpStream.eof())
        {
                ExpStream >> TestString >> test;
                if (TestString.length()>1)
       {
              NewNode = new ProbeNode;
           NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->SynthesizeMutant(MutateValue);
        this->DataSet.InsertAfterCurrent(NewNode);
                }
        }
        ExpStream.close();
   cout << DataSet.Cost << endl;
}

TSPClass::LoadMisMatchData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
   string TestString;
        ProbeNode *NewNode;
        float test; // soak up number

        while (!ExpStream.eof())
        {
```

```
                ExpStream >> TestString >> test;
                if (TestString.length()>1)
                {
                        NewNode = new ProbeNode;
           NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->SynthesizeMisMatch(MutateValue);
                        this->DataSet.InsertAfterCurrent(NewNode);
      }
        }
        ExpStream.close();
   cout << DataSet.Cost << endl;
}

TSPClass::LoadMisMatchFluffData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
   string TestString;
        ProbeNode *NewNode;
        float test; // soak up number

        while (!ExpStream.eof())
        {
                ExpStream >> TestString >> test;
      if (TestString.length()>1)
                {
             NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Probe = TestString;
           NewNode->DataPointer->SynthesizeFluff(MutateValue);
        this->DataSet.InsertAfterCurrent(NewNode);
        }
        }
        ExpStream.close();
   cout << DataSet.Cost << endl;
}

TSPClass::LoadMisMatchFastFluffData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
   string TestString;
        ProbeNode *NewNode;
        float test; // soak up number

        while (!ExpStream.eof())
        {
                ExpStream >> TestString >> test;
      if (TestString.length()>1)
                {
             NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
```

```
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->SynthesizeFastFluff(MutateValue);
                        this->DataSet.InsertAfterCurrent(NewNode);
                }
                }
                ExpStream.close();
                cout << DataSet.Cost << endl;
}

TSPClass::LoadExpressionData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
        string TestString;
        ProbeNode *NewNode;
        float test; // soak up number
        long facevalue;
        long Unit, Atom;
        string Name, Rename;
        string Junk;
        long counter = 0;
        ExpStream >> Junk >> Junk >> Junk >> Junk >> Junk >> Junk;
        while (!ExpStream.eof())
        {
                ExpStream >> facevalue >> TestString >> Name >> Rename >> Unit >> Atom;
                if (TestString.length()>1)
                {
                        NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Original = TestString;
                        transform(TestString);
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->Name = Name;
                        NewNode->DataPointer->Rename = Rename;
                        NewNode->DataPointer->Location = facevalue;
                        NewNode->DataPointer->Unit = Unit;
                        NewNode->DataPointer->Atom = Atom;
                        NewNode->DataPointer->SynthesizeMisMatch(MutateValue);
                        this->DataSet.InsertAfterCurrent(NewNode);
                }
                counter++;
                if (counter%100==0)
                        cout << counter << TAB << Name << TAB << TestString << endl;
        }
        ExpStream.close();
        cout << DataSet.Cost << endl;
}
TSPClass::LoadSingleExpressionData(string FileName)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        int probeloop = 0;
        string TestString;
        ProbeNode *NewNode;
```

```
        float test; // soak up number
        int facevalue;
        int Unit, Atom;
        string Name;
        string Junk;
        int counter = 0;
        ExpStream >> Junk >> Junk >> Junk >> Junk >> Junk >> Junk;
        while (!ExpStream.eof())
        {
                ExpStream >> facevalue >> TestString >> Name >> Junk >> Unit >> Atom;
                if (TestString.length()>1)
                {
                        NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Original = TestString;
                        transform(TestString);
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->Name = Name;
                        NewNode->DataPointer->Location = facevalue;
                        NewNode->DataPointer->Unit = Unit;
                        NewNode->DataPointer->Atom = Atom;
                        NewNode->DataPointer->Synthesize();
                        this->DataSet.InsertAfterCurrent(NewNode);
                }
                counter++;
                if (counter%100==0)
                        cout << counter << TAB << Name << TAB << TestString << endl;
        }
        ExpStream.close();
        cout << DataSet.Cost << endl;
}

TSPClass::LoadExpressionDataByUnit(string FileName, long MutateValue, long UnitLimit)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
        string TestString;
        ProbeNode *NewNode;
        float test; // soak up number
        long facevalue;
        long Unit, Atom;
        string Name;
        string Junk;
        long counter = 0;
        while (!ExpStream.eof())
        {
                ExpStream >> facevalue >> TestString >> Name >> Junk >> Unit >> Atom;
                if (TestString.length()>1 && Unit==UnitLimit)
                {
                        NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Original = TestString;
                        transform(TestString);
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->Name = Name;
                        NewNode->DataPointer->Location = facevalue;
                        NewNode->DataPointer->Unit = Unit;
                        NewNode->DataPointer->Atom = Atom;
                        NewNode->DataPointer->SynthesizeMisMatch(MutateValue);
                        this->DataSet.InsertAfterCurrent(NewNode);
                }
                counter++;
```

```
      if (counter%100==0)
        cout << counter << TAB << Name << TAB << TestString << endl;
        }
        ExpStream.close();
        cout << DataSet.Cost << endl;
}

TSPClass::LoadExpressionFluffData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
        string TestString;
        ProbeNode *NewNode;
        float test; // soak up number
        long facevalue;
        long Unit, Atom;
        string Junk;
        string Name;
        long counter = 0;
        while (!ExpStream.eof())
        {
        ExpStream >> facevalue >> TestString >> Name;
      if (TestString.length()>1)
              {
                      NewNode = new ProbeNode;
         NewNode->DataPointer = new ProbeSynth;
                      transform(TestString);
            NewNode->DataPointer->Probe = TestString;
          NewNode->DataPointer->Name = Name;
                      NewNode->DataPointer->Location = facevalue;
                      NewNode->DataPointer->SynthesizeFastFluff(MutateValue);
                      this->DataSet.InsertAfterCurrent(NewNode);
              }
      counter++;
      if (counter%100==0)
        cout << counter << TAB << Name << TAB << TestString << endl;
        }
        ExpStream.close();
        cout << DataSet.Cost << endl;
}

TSPClass::LoadChipData(string FileName)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
        string TestString;
   ProbeNode *NewNode;
        long test; // soak up number

        while (!ExpStream.eof())
        {
                ExpStream >> TestString >> test;
      if (TestString.length()>1)
```

```
        {
                NewNode = new ProbeNode;
            NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->SynthesizeMutant(test-1);
                        this->DataSet.InsertAfterCurrent(NewNode);
                }
        }
        ExpStream.close();
        cout << DataSet.Cost << endl;
}

TSPClass::OutputTour(string FileName)
{
        ofstream OutStream;
        OutStream.open(FileName.c_str(), ios::out);

        this->BestTour.Output(OutStream);

        OutStream.close();
}

TSPClass::ListTour(string FileName)
{
        ofstream OutStream;
        OutStream.open(FileName.c_str(), ios::out);

        this->BestTour.List(OutStream);

        OutStream.close();
}

TSPClass::AppendTour(string FileName)
{
        ofstream OutStream;
        OutStream.open(FileName.c_str(), ios::app);

        this->BestTour.Output(OutStream);

        OutStream.close();
}

static void do_gematch(long Value)
{
        TSPClass Example;

        Example.LoadMisMatchFluffData("gematch.prb", 10-1);
        Example.GenerateTourByInsertionWithModifier(Value);
        Example.OutputTour("gematch.flf");
}

static void do_yematch()
{
        TSPClass Example;

        Example.LoadExpressionData("yematch.prb", 10-1);
        Example.GenerateTourByInsertionWithModifier(1);
        Example.OutputTour("yematch.tsp");
}

static void do_yematch_local(long UnitMatch)
{
        TSPClass Example;

        Example.LoadExpressionDataByUnit("yematch.prb", 10-1, UnitMatch);
        Example.GenerateTourByInsertionWithModifier(1);
        Example.AppendTour("yematch.lsp");
}

static void do_yematch_local_pool(long UnitMatch)
{
```

```
        TSPClass Example;

        Example.LoadExpressionDataByUnit("yematch.prb", 10-1, UnitMatch);
        Example.GenerateTourByClosestPool();
        Example.AppendTour("yematch.csp");
}

static void do_hummatch_local(long UnitMatch)
{
        TSPClass Example;

        Example.LoadExpressionData("ha.prb", 13-1);
        Example.GenerateQuickTourByInsertion(2048);
        Example.OutputTour("ha.tsp");

        //while (Example.GenerateTourByInsertionAndDeletion())
                //Example.AppendTour("ha.lsp");
}

static void do_fast_gematch(long Value)
{
        TSPClass Example;

        Example.LoadMisMatchFastFluffData("gematch.prb", 10-1);
        Example.GenerateTourByInsertionWithModifier(Value);
        Example.OutputTour("gefast.flf");
}

static void do_fast_pool_gematch(long Value)
{
        TSPClass Example;

        Example.LoadMisMatchFastFluffData("gematch.tny", 10-1);
        Example.GenerateTourByClosestPool();
        Example.OutputTour("gefast.pl");
}

static void do_mito(long Value)
{
        TSPClass Example;

        Example.LoadMutantData("mt9566.prb", 10-1);
        Example.GenerateTourByInsertionWithModifier(Value);
        //Example.ImproveTourByReplacement(1000);
        string Test;
        Test = "mt9566.";
   char ctest = 'a'+Value;
        Test += ctest;
   Example.OutputTour(Test);
}


static void do_hiv(long Value)
{
        TSPClass Example;

        Example.LoadChipData("hv430a.prb");
        Example.GenerateTourByInsertionWithModifier(Value);
        //Example.ImproveTourByReplacement(1000);
        string Test;
   Test = "hv430.";
        char ctest = 'a'+Value;
   Test += ctest;
        Example.OutputTour(Test);
}

static void do_new_ge()
{
        TSPClass Example;
```

```
        Example.LoadExpressionFluffData("i:\\data\\metrix\\ehubbe\\design\\nmisc\\gem01\\gem01.prb",
10-1);
        Example.GenerateTourByInsertionWithModifier(1);
        Example.OutputTour("gem01.flf");
}


static void do_noise()
{
        TSPClass Example;

        Example.LoadData("noisea8.20");
        Example.GenerateTourByInsertionWithModifier(1);
        Example.ListTour("na8_20.tsp");
}

static void do_noise_two()
{
        TSPClass Example;

        Example.LoadData("c:\\cover\\noisea8.16");
        Example.GenerateTourByInsertionWithModifier(1);
        Example.ListTour("na8_16.tsp");
}

static void do_noise_three()
{
        TSPClass Example;

        Example.LoadData("c:\\cover\\noisea8.18");
        Example.GenerateTourByInsertionWithModifier(1);
        Example.ListTour("na8_18.tsp");
}

static void do_cost_one()
{
        TSPClass Example;

        Example.LoadData("c:\\genius\\testa8.20");
        Example.GenerateTourByInsertionWithModifier(1);
        Example.ListTour("ca8_20.tsp");
}

static void do_cost_two()
{
        TSPClass Example;

        Example.LoadData("c:\\genius\\ca8_20.prb");
        Example.GenerateTourByInsertionWithModifier(1);
        Example.ListTour("cba8_20.tsp");
}

static void do_cost_quick()
{
        TSPClass Example;

        Example.LoadData("c:\\genius\\noise\\ca8_20.rnd");
        Example.GenerateQuickTourByInsertion(1024);
        Example.ListTour("cqa8_20.45");
}

static void do_rat_local(long UnitMatch)
{
        TSPClass Example;

        Example.LoadExpressionData("r:\\alldes\\cdesign\\ter09\\included_probes\\normal_probes.txt",
13 -1 );
        Example.LoadExpressionData("r:\\alldes\\cdesign\\ter09\\included_probes\\sense\\reverse_comp
_seqs.txt", 13 -1);
//      Example.LoadSingleExpressionData("r:\\alldes\\cdesign\\eol0191\\eoshu02.dat");
//      Example.GenerateQuickTourByInsertion(140000);
```

```
        Example.GenerateQuickTourByInsertion(20480);
        Example.OutputTour("r:\\alldes\\cdesign\\ter09\\full_fix.tsp");

        //while (Example.GenerateTourByInsertionAndDeletion())
                //Example.AppendTour("ha.lsp");
}

main()
{
        do_rat_local(0);
//      do_fast_pool_gematch(1);
//      do_yematch();
//      for (long i=1; i<104; i++)
//              do_yematch_local_pool(i);
//      do_gematch(1);
//    do_new_ge();
//      do_noise();
//      do_noise_two();
//      do_noise_three();
//      do_cost_quick();
}
```

Appendix B

Edgeopt.cpp

Attorney Docket 3296.1

Inventor: Earl A. Hubbell


```
// Edge Optimizer
#include <cstring.h>
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>
#include <time.h>
#include <math.h>

#define TAB '\t'
#define MXNAME 40
#define MXLINE 1000

#define TRUE 1
#define FALSE 0
#define MXSEQ 45
#define MXFEATURE 45
#define MXQUALIFIER 45

// Edge Optimizer Story
// 1) Strip off all Valid Blocks
// 2) Put Valid Blocks On to Minimize Edges

// Input: Cdl file, Ret file, Parameters
// Output: Twisted Cdl file and Ret file

char complement(char base)
{
        if (base=='A')
        return('T');
    if (base=='C')
        return('G');
    if (base=='G')
        return('C');
    if (base=='T')
        return('A');
    return(base);
}


class EntryClass{
// what CDL information is associated with everything
public:
        char sequence[MXSEQ];
        int destype;
        char feature[MXFEATURE];
```

```
        char qualifier[MXQUALIFIER];
        int expos;
        int     endpos;
        int pos;
        char pbase[MXFEATURE], tbase[MXFEATURE];
    int finishpos;
    int fixed;
    int variable;
        int unit, block;
    long atom;
    int repeat;
    int seqno;
    long layout;
        char locus[MXFEATURE];
    char accession[MXFEATURE];
    EntryClass(){Initialize();};
    Initialize();
    LineScan(char *);
    DumpLine(FILE *fp, int i, int j);
    DumpMut(FILE *fp);
};


EntryClass::Initialize()
{
        strcpy(sequence, "");
        destype = 0;
        strcpy(feature, "");
        strcpy(qualifier, "");
    expos = 0;
    pos = 0;
    strcpy(pbase,"!");
    strcpy(tbase,"!");
    unit = 0;
    block = 0;
    atom = 0;
}

EntryClass::LineScan(char *Line)
{
        int X,Y;
        static  char    PROBE[MXLINE];
        int     DESTYPE;
        static  char    FEATURE[MXLINE],
                        QUALIFIER[MXLINE];
        int     EXPOS;
        char    TBASE[MXLINE];
        int     ENDPOS,
                        POSITION;
        char  PBASE[MXLINE];
        int     FINISHPOS,
                        FIXED,
                        VARIABLE,
                        UNIT,
                        BLOCK,
                        REPEAT,
                        SEQNO,
                        LAYOUT;
    long ATOM;
        static  char  ACCESSION[MXLINE],
                        LOCUS[MXLINE];

                        sscanf(Line, "%d %d %s %d %s %s %d %s %d %d %s %d %d %d %d %d %ld %d %d %d %s
%s",
                                                                        &X,
                                                                        &Y,
                                                                        PROBE,
                                                                        &DESTYPE,
                                                                        FEATURE,
                                                                        QUALIFIER,
                                                                        &EXPOS,
```

```
                                                                TBASE,
                                                                &ENDPOS,
                                                                &POSITION,
                                                                PBASE,
                                                                &FINISHPOS,
                                                                &FIXED,
                                                                &VARIABLE,
                                                                &UNIT,
                                                                &BLOCK,
                                                                &ATOM,
                                                                &REPEAT,
                                                                &SEQNO,
                                                                &LAYOUT,
                                                                LOCUS,
                                                                ACCESSION);
                strcpy(sequence, PROBE);
                destype = DESTYPE;
                strcpy(feature, FEATURE);
                strcpy(qualifier, QUALIFIER);
                strcpy(locus, LOCUS);
                strcpy(accession, ACCESSION);
                expos = EXPOS;
                endpos = ENDPOS;
                finishpos = FINISHPOS;
        fixed = FIXED;
        variable = VARIABLE;
                repeat = REPEAT;
                seqno = SEQNO;
                layout = LAYOUT;

                strcpy(tbase, TBASE);
                strcpy(pbase, PBASE);
                unit = UNIT;
                block = BLOCK;
                atom = ATOM;
                pos = POSITION;
}

EntryClass::DumpLine(FILE *fp, int i, int j)
{
        fprintf(fp,
"%d\t%d\t%s\t%d\t%s\t%s\t%d\t%s\t%d\t%d\t%s\t%d\t%d\t%d\t%d\t%d\t%ld\t%d\t%d\t%d\t%s\t%s\n",
                                                                i,
                                                                j,
                                                                sequence,
                                                                destype,
                                                                feature,
                                                                qualifier,
                                                                expos,
                                                                tbase,
                                                                endpos,
                                                                pos,
                                                                pbase,
                                                                finishpos,
                                                                fixed,
                                                                variable,
                                                                unit,
                                                                block,
                                                                atom,
                                                                repeat,
                                                                seqno,
                                                                layout,
                                                                locus,
                                                                accession);
}

EntryClass::DumpMut(FILE *fp)
{
char tempc;

        if (destype==0)
```

```
    {
        fprintf(fp, "~");
        return(TRUE);
    }
    if (abs(destype)<100)
    {
        if (destype>0)
                fprintf(fp, "C");
        else
                fprintf(fp, "X");
        return(TRUE);
    }
        if (strlen(pbase)>1)
    {
        fprintf(fp, "I");
        return(TRUE);
    }
    if (pbase[0]=='!')
    {
        fprintf(fp, "D");
        return(TRUE);
    }
    if (destype>0)
    {
        tempc = complement(pbase[0]);
    }
    else
        tempc = pbase[0];
    if (tempc==tbase[0])
    {
                fprintf(fp, "%c", tempc);
        return(TRUE);
    }
    fprintf(fp, " ");
    return(TRUE);
}


class SynthClass{
// how things are built
public:
    char *synthesis;
    int synlength;
        SynthClass(){synthesis=NULL; synlength = 0;};
    Allocate(int);
    DeAllocate();
    Diff(SynthClass &);
    SetBit(char, int);
    char GetBit(int);
    GetLast();
    GetFirst();
    ~SynthClass();
};

SynthClass::~SynthClass()
{
        DeAllocate();
}

SynthClass::Allocate(int Size)
{
        synthesis = new char [Size]; // just a bitfield, really
    if (synthesis==NULL)
    {
        printf("Blow up! In SynthClass::Allocate");
        return(FALSE);
    }
    for (int i=0; i<Size; i++)
        synthesis[i] = 0; // Nothing, null, no bits set!
    synlength = Size;
        return(TRUE);
```

```
}

SynthClass::DeAllocate()
{
        if (synthesis!=NULL)
                delete[] synthesis;
    synlength = 0;
    synthesis = NULL;
}

SynthClass::SetBit(char value, int Which)
{
        if (synthesis!=NULL && Which<synlength && Which>-1)
                synthesis[Which] = value;
}

char
SynthClass::GetBit(int Which)
{
        if (synthesis!=NULL && Which<synlength && Which>-1)
                return(synthesis[Which]);
    else
        return(0);
}

SynthClass::Diff(SynthClass &Source)
{
        int count = 0;
    if (!(synthesis!=NULL && Source.synthesis!=NULL && synlength==Source.synlength))
        return(0);
        for (int i=0; i<synlength; i++)
    {
                count += (synthesis[i]!=Source.synthesis[i]);
    }
    return(count);
}

SynthClass::GetLast()
{
        for (int i=synlength-1; i>0; i--)
        if (synthesis[i]>0)
                    return(i);
}

SynthClass::GetFirst()
{
        for (int i=0; i<synlength; i++)
        if (synthesis[i]>0)
                    return(i);
}

class LocalDataClass{
public:
        int relx, rely;  // relative positions within a block
    int validflag; // newly added - can I move this for optimization?
        EntryClass cdldata;
    SynthClass retdata;
    LocalDataClass(){relx=rely=0;validflag = TRUE;};
    ~LocalDataClass();
    SetRelative(int, int);
    PrintLongSeq(FILE *fp);
};

LocalDataClass::~LocalDataClass()
{
        retdata.DeAllocate();
}

LocalDataClass::SetRelative(int i, int j)
{
        relx = i;
```

```
    rely = j;
}

LocalDataClass::PrintLongSeq(FILE *fp)
{
        int j, i=3;

        for (j=0; j<retdata.synlength; j++)
    {
        if (retdata.GetBit(j))
        {
                fprintf(fp, "%c", cdldata.sequence[i]);
            i++;
        }
        else
                fprintf(fp, ".");
    }
}



class BlockClass{
public:
        LocalDataClass **DataStack;
    int DataStackSize;
    int WSize, HSize; // rectangular grid
    BlockClass(){DataStack=NULL; DataStackSize = 0; WSize = HSize = 0;};
    Allocate(int);
    DeAllocate();
    ~BlockClass();
};

BlockClass::Allocate(int Size)
{
        DataStack = new LocalDataClass *[Size];
    if (DataStack==NULL)
        return(FALSE);
    for (int i=0; i<Size; i++)
        DataStack[i]=NULL;
    DataStackSize = Size;
}

BlockClass::DeAllocate()
{
        if (DataStack==NULL)
        return(TRUE);
    for (int i=0; i<DataStackSize; i++)
        if (DataStack[i]!=NULL)
        {
                printf("Error!  Data Leakage from Block!\n");
                        delete DataStack[i];
            DataStack[i] = NULL;
        }
    delete[] DataStack;
    DataStack = NULL;
    DataStackSize = 0;
    WSize = HSize = 0;
}

BlockClass::~BlockClass()
{
        DeAllocate();
}

class BlockStackClass{
public:
        BlockClass **BlockStack;
    long BlockCurSize;
    long BlockStackSize;
    BlockStackClass(){BlockStack=NULL; BlockStackSize = 0;BlockCurSize = 0;};
    Allocate(long);
```

```
    DeAllocate();
    ~BlockStackClass(){DeAllocate();};
    BlockClass *PutBlockOnStack(BlockClass *);
    BlockClass *RemoveBlock(long);
    BlockClass *TemporaryBlockFromStack(long);
    Swap(long, long);
    Shuffle();
};

BlockStackClass::Allocate(long Size)
{
        BlockClass ** TmpStack;

        TmpStack = new BlockClass * [Size];
    if (TmpStack==NULL)
        return(FALSE);
    long i;

    for (i=0; i<Size; i++)
        TmpStack[i]=NULL;
    for (i=0; i<BlockCurSize && i<BlockStackSize && BlockStack!=NULL; i++)
    {
        TmpStack[i] = BlockStack[i];
         BlockStack[i] = NULL;
    }
    if (BlockStack!=NULL)
        delete[] BlockStack;
    BlockStack = TmpStack;
    TmpStack = NULL;
    BlockStackSize = Size;
    return(TRUE);
}

BlockClass *
BlockStackClass::PutBlockOnStack(BlockClass *TempBlock)
{
        if (BlockCurSize<BlockStackSize)
    {
    }
    else
    {
        if (!Allocate(BlockStackSize+1000))
        {
                printf("Can't increase block stack\n");
                return(TempBlock); // upgrade stack
        }
    }
    BlockStack[BlockCurSize] = TempBlock;
    BlockCurSize++;
    return(NULL); // remove pointer
}

BlockStackClass::Swap(long Source, long Sink)
{
        BlockClass *TempBlock;

    TempBlock = BlockStack[Sink];
    BlockStack[Sink] = BlockStack[Source];
    BlockStack[Source] = TempBlock;
    TempBlock = NULL;
    return(TRUE);
}

BlockClass *
BlockStackClass::RemoveBlock(long WhichBlock)
{
        BlockClass *TempBlock;

    if (WhichBlock>=BlockCurSize || WhichBlock<0)
        return(NULL);
```

```
        TempBlock = BlockStack[WhichBlock];
    BlockCurSize--;
    BlockStack[WhichBlock] = BlockStack[BlockCurSize];
    BlockStack[BlockCurSize]=NULL;
    return(TempBlock);
}

BlockClass *
BlockStackClass::TemporaryBlockFromStack(long WhichBlock)
{
        BlockClass *TempBlock;

    if (WhichBlock>=BlockCurSize || WhichBlock<0)
        return(NULL);

        TempBlock = BlockStack[WhichBlock];
    return(TempBlock);
}

BlockStackClass::DeAllocate()
{
        if (BlockStack==NULL)
        return(TRUE);
    long i;
    for (i=0; i<BlockStackSize; i++)
    {
        if (BlockStack[i]!=NULL)
        {
                printf("Data Leakage from BlockStack\n");
            delete BlockStack[i];
            BlockStack[i] = NULL;
        }
    }
    delete[] BlockStack;
    BlockStack = NULL;
    BlockStackSize = 0;
    BlockCurSize = 0;
}

BlockStackClass::Shuffle()
{
        long t;
    long i;

        // randomly rearrange the stack to prevent bias
    for (i=BlockCurSize-1; i>0; i--)
    {
        t = rand()%32000;
        t = t*32000+(rand()%32000);
        t = t %(i+1);
                Swap(i,t);
    }
}

class ChipArrayClass{
public:
        BlockStackClass ValidBlockStack;
    LocalDataClass ***DataGrid;
    int Xdim;
    int Ydim;
    int SynthSteps;

    char retname[MXNAME];
    long NumOnes; // useful statistic

    int GlobalHeight;
    int MaxAllowed;
    int Radius;

    float LeakageHalfLife;
    int ScanRadius;
```

```
    int weightflag; // type of weights to use

    // constraints


    ChipArrayClass();
    ~ChipArrayClass();
    Allocate(int, int);
    DeAllocate();
    ReadCdl(char *, int);
    DumpCdl(char *);
    ReadRet(char *, int);
    DumpRet(char *);

    StripAreaToBlock(BlockClass *, int,int,int,int);
    CheckBlockFitToArea(BlockClass *, int, int);
        PutBlockInArea(BlockClass *, int, int);

    ValidMove(int, int);
    ValidLocation(int, int);
    Valid(int,int);
    ValidBlock(int,int,int,int);
    ValidTile(int,int,int,int);
    ValidBlank(int,int,int,int);

    CountDiff(LocalDataClass *,int,int);
    double CountEdges(BlockClass *, int, int);
    double CountWeightedEdges(BlockClass *, int, int);
    double CountEdgesFromStack(long, int, int);
    FindNextDiagonalSlot(int &, int &);
    FindNextHorizontalSlot(int &, int &);
    DiagonalReplacement(long);
    HorizontalReplacement(long);
    StripAllValidBlocks(int);
    PlaceBlockFromStack(long, int,int);
    SearchLocationWithStats(int,int,long, long &, double &, double &, double &);

    CountUnitInArea(long, int, int,int,int);
    ProximityCheckBlock(BlockClass *, int,int,int,int);
    ProximityCheckFromStack(long, int,int,int,int);

    StripBadProximityValues(int);
    PickRandomValidBlock(int &, int &, int, int);

    ReadInstructionFile(char *);
    InterpretInstructionLine(char *);
    GenerateMutFile(char *);
    SetUnits(long, long, int);
    SetArea(int, int, int, int, int);
    SetAntiArea(int,int,int,int,int);
    SetDestype(int, int);
    Shuffle();
    StripValidBlock(int,int,int);
    StripRandomBlocks(long, int);
    DoubleReplacement(long);
    GenerateDiffFile(char *);

    FindNextAggregateSlot(int &, int &);
    AggregateReplacement(long);
};

ChipArrayClass::Shuffle()
{
        ValidBlockStack.Shuffle();
}

ChipArrayClass::SetArea(int X, int Y, int tX, int tY, int value)
{
        int i,j;
        for (i=X; i<=tX; i++)
    {
```

```
        for (j=Y; j<=tY; j++)
        {
                if (Valid(i,j))
                        DataGrid[i][j]->validflag = value;
        }
    }
}

ChipArrayClass::SetAntiArea(int X, int Y, int tX, int tY, int value)
{
        // used to set all >but< a given physical area
    int i,j;
    for (i=0; i<Xdim; i++)
    {
        for (j=0; j<Ydim; j++)
        {
                if (Valid(i,j))
            {
                if (!(i>=X && i<=tX && j>=Y && j<=tY))
                        DataGrid[i][j]->validflag = value;
            }
        }
    }
}

ChipArrayClass::SetDestype(int destype, int value)
{
        int i,j;

        for (i=0; i<Xdim; i++)
    {
        for (j=0; j<Ydim; j++)
        {
                if (Valid(i,j))
            {
                if (destype==DataGrid[i][j]->cdldata.destype)
                 {
                        DataGrid[i][j]->validflag = value; // can't move this one
                 }
            }
        }
    }

}

ChipArrayClass::ValidMove(int X, int Y)
{
        if (DataGrid[X][Y]->validflag)
        return(TRUE);
    return(FALSE);
}

ChipArrayClass::ValidLocation(int X, int Y)
{
        if (X<0 || Y<0 || X>=Xdim || Y>=Ydim)
        return(FALSE);
        // nothing here yet
    return(TRUE);
}

ChipArrayClass::Valid(int X, int Y)
{
    if (!ValidLocation(X,Y))
        return(FALSE); // not a location we're allowed to move
    if (DataGrid[X][Y]==NULL)
        return(FALSE); // doesn't even exist!
    return(TRUE);
}

ChipArrayClass::SetUnits(long Start, long Finish, int value)
{
```

```
        int i,j;

        for (i=0; i<Xdim; i++)
    {
        for (j=0; j<Ydim; j++)
        {
                if (Valid(i,j))
            {
                if (Start<=DataGrid[i][j]->cdldata.unit && Finish>=DataGrid[i][j]->cdldata.unit)
                {
                        DataGrid[i][j]->validflag = value; // can't move this one
                }
            }
        }
    }
}


ChipArrayClass::ValidTile(int X, int Y, int Width, int Height)
{
        long U, A;
    int tx, ty,i,j;
    int xl, yl;
    xl = X+Width;
    yl = Y+Height;

        U = DataGrid[X][Y]->cdldata.unit;
    A = DataGrid[X][Y]->cdldata.atom;
//    if (Width==Height && Height==1)  // if we're moving control probes around!
//            return(TRUE);
        for (i=-1; i<Width+1; i++)
    {
        for (j=-1; j<Height+1; j++)
        {
                        tx = i+X;
            ty = j+Y;
            if (tx>=X && ty>=Y && tx<xl && ty<yl)
            {
                if (!Valid(tx,ty) || !ValidMove(tx,ty))   // if not valid we're unhappy
                        return(FALSE);
                 if (U!=DataGrid[tx][ty]->cdldata.unit || A!=DataGrid[tx][ty]->cdldata.atom) // if
not same, we're unhappy
                        return(FALSE);
            }
            else
            {
                if (Valid(tx,ty))
                {
                        if (U==DataGrid[tx][ty]->cdldata.unit && A==DataGrid[tx][ty]->cdldata.atom)
// if outside same block unhappy
                        return(FALSE);
                }
            }
        }
    }
    return(TRUE);
}

ChipArrayClass::ValidBlank(int X, int Y, int Width, int Height)
{
        int i,j, tx,ty;

        // valid set of blanks
    for (i=0; i<Width; i++)
        for (j=0; j<Height; j++)
        {
                tx = i+X;
            ty = j+Y;
            if (!Valid(tx,ty) || !ValidMove(tx,ty))
                return(FALSE);
            if (DataGrid[tx][ty]->cdldata.destype!=0)
```

```
                return(FALSE);
            }
        return(TRUE);
}

ChipArrayClass::ValidBlock(int X, int Y, int Width, int Height)
{

        if (!Valid(X,Y))
        return(FALSE);
    if (!ValidMove(X,Y))
        return(FALSE);
    if (ValidTile(X,Y,Width, Height))
        return(TRUE);
    if (ValidBlank(X,Y, Width, Height)) // allow blank blocks to be moved, if validflag set for
destype 0
        return(TRUE);

    return(FALSE);
}

ChipArrayClass::PickRandomValidBlock(int &X, int &Y, int Width, int Height)
{
        long counter = 0;
    long Limit = 10000;
        X = rand()%Xdim;
    Y = rand()%Ydim;
        while(!ValidBlock(X,Y, Width, Height) && counter<Limit)
    {
        X = rand()%Xdim;
         Y = rand()%Ydim;
         counter++;
    }
    if (counter==Limit)
        return(FALSE); // can't find one in reasonable time!
    return(TRUE);
}


ChipArrayClass::CountDiff(LocalDataClass *TestData, int X, int Y)
{
        if (!Valid(X,Y))
        return(0);  // doesn't exist or is off chip, so no problems!
    return(TestData->retdata.Diff(DataGrid[X][Y]->retdata));
}

double
ChipArrayClass::CountEdges(BlockClass *TempBlock, int X, int Y)
{
        int i,tx,ty;
    int count = 0;
        // count the edges, if this block is in this location
    // note that "interior" edges of blocks are >not< counted
        for (i=0; i<TempBlock->DataStackSize; i++)
    {
        if (TempBlock->DataStack[i]!=NULL)
        {
                tx = X+TempBlock->DataStack[i]->relx;
                ty = Y+TempBlock->DataStack[i]->rely;
            count +=CountDiff(TempBlock->DataStack[i], tx+1, ty);
            count +=CountDiff(TempBlock->DataStack[i], tx-1, ty);
            count +=CountDiff(TempBlock->DataStack[i], tx, ty-1);
            count +=CountDiff(TempBlock->DataStack[i], tx, ty+1);
        }
    }
    return(count);
}

double
ChipArrayClass::CountWeightedEdges(BlockClass *TempBlock, int X, int Y)
{
```

```
    int i,tx,ty;
    int rangex,rangey;
    double count = 0;
    double lcount;
    double distance;
    // count the edges, if this block is in this location
    // note that "interior" edges of blocks are >not< counted
    for (i=0; i<TempBlock->DataStackSize; i++)
    {
    if (TempBlock->DataStack[i]!=NULL)
    {
        for (rangex = -1*ScanRadius; rangex<=ScanRadius; rangex++)
        {
        for (rangey=-1*ScanRadius; rangey<=ScanRadius; rangey++)
        {
            if (rangex!=0 || rangey!=0)
            {
                distance = (rangex*rangex)+(rangey*rangey);
                distance = sqrt(distance);
                    tx = X+TempBlock->DataStack[i]->relx+rangex;
                ty = Y+TempBlock->DataStack[i]->rely+rangey;
                lcount =CountDiff(TempBlock->DataStack[i], tx, ty);
                lcount*=pow(.5,LeakageHalfLife*distance);
                count +=lcount;
            }
        }
        }
    }
    }
    return(count);
}

ChipArrayClass::CountUnitInArea(long Unit, int X, int Y, int Width, int Height)
{
    int i,j;
    int tx, ty;
    int count =0;
    long tatom=-100; // unlikely in any real unit

    for (i=0; i<Width; i++)
    {
    tatom = -100; // start over with each vertical stripe - assumes vertical "atoms"
    for (j=0; j<Height; j++)
    {
        tx = X+i;
        ty= Y+j;
        if (Valid(tx,ty))
        {
        if (DataGrid[tx][ty]->cdldata.unit==Unit)
        {
            // works because we're scanning vertically
            if (tatom!=DataGrid[tx][ty]->cdldata.atom) // only count a given unit/atom
once
                count++;
            tatom = DataGrid[tx][ty]->cdldata.atom;
        }
        }
    }
    }
    return(count);
}

ChipArrayClass::ProximityCheckBlock(BlockClass *TempBlock, int X, int Y, int Width, int Height)
{
    return(CountUnitInArea(TempBlock->DataStack[0]->cdldata.unit,X,Y,Width,Height));
}

ChipArrayClass::ProximityCheckFromStack(long Which, int X, int Y, int Width, int Height)
{
    BlockClass *TempBlock;
```

```
    int count;

    TempBlock = ValidBlockStack.TemporaryBlockFromStack(Which);
    if (TempBlock==NULL)
        return(16000); // big error
    count = ProximityCheckBlock(TempBlock, X, Y,Width, Height);
    TempBlock = NULL;
    return(count);
}

double
ChipArrayClass::CountEdgesFromStack(long Which, int X, int Y)
{
        BlockClass *TempBlock;
    double count;

    TempBlock = ValidBlockStack.TemporaryBlockFromStack(Which);
    if (TempBlock==NULL)
        return(16000); // big error
    if (!weightflag)
            count = CountEdges(TempBlock, X, Y);
    else
        count = CountWeightedEdges(TempBlock, X,Y);
    TempBlock = NULL;
    return(count);
}

ChipArrayClass::FindNextDiagonalSlot(int &X, int &Y)
{
        // look for a NULL entry
    while (Y<Ydim && DataGrid[X][Y]!=NULL)
    {
        X-=1;
        Y+=1;
        if (X<0 || Y>=Ydim)
        {
                X = Y+X+1;
            Y = 0;
        }
        if (X>=Xdim)
        {
                Y = X-(Xdim-1);
            X = Xdim-1;
        }
    }
    if (Y==Ydim) // ran out of room!
        return(FALSE);
    else
        return(TRUE);
}

ChipArrayClass::FindNextHorizontalSlot(int &X, int &Y)
{
        // look for a NULL entry
    while (Y<Ydim && DataGrid[X][Y]!=NULL)
    {
        X +=1;
        if (X>=Xdim)
        {
                X = 0;
            Y++;
        }
    }
    if (Y==Ydim) // ran out of room!
        return(FALSE);
    else
        return(TRUE);
}

ChipArrayClass::FindNextAggregateSlot(int &X, int &Y)
{
```

```
        int total;
        total = X;
    if (Y>total)
        total = Y;
    while (Y<Ydim && DataGrid[X][Y]!=NULL)
    {
        // look for slots
        if (X>Y)
               Y++; // move vertically
        else
                if (X<=Y)
                      X--;  // basic zero moves
        if (X<0)
        {
               X=Y+1; // add one to total
            Y=0;
        }
        if (X>=Xdim)
        {
               Y = X;
            X=Xdim-1;
        }
    }
    if (Y>=Ydim)
        return(FALSE);
    else
            return(TRUE);
}

ChipArrayClass::PlaceBlockFromStack(long Which, int X, int Y)
{
       BlockClass *TempBlock;

              TempBlock = ValidBlockStack.RemoveBlock(Which);
            if (TempBlock!=NULL)
            {
                            if (PutBlockInArea(TempBlock, X,Y))
                     TempBlock = NULL; // keep wacky pointers from drifting
                 else
                 {
                       printf("Failure to fit block in area: %d %d\n", X, Y);
                    TempBlock = ValidBlockStack.PutBlockOnStack(TempBlock); // throw back on stack
                 }
            }
            else
               printf("Failure to get from stack! %d %d\n", X,Y);
}

ChipArrayClass::SearchLocationWithStats(int X, int Y, long searchlimit, long &Best, double &bestc,
double &avg, double &worstc)
{
       long search;
    long count = 0;
    double c;

        Best = ValidBlockStack.BlockCurSize-1; // which one
        bestc = CountEdgesFromStack(ValidBlockStack.BlockCurSize-1,X,Y);
               avg = bestc;
        worstc = bestc;
        count = 1;
        for (search=1; search<searchlimit && search<ValidBlockStack.BlockCurSize; search++)
        {
                       c = CountEdgesFromStack(ValidBlockStack.BlockCurSize-1-search,X,Y); // what
if we put here?
            avg +=c;
            if (c<bestc)
            {
               bestc = c;
                Best = ValidBlockStack.BlockCurSize-1-search;
            }
            if (c>worstc)
```

```
            {
               worstc=c;
            }
            count++;
        }
        avg /=count; // number actually searched
        return(TRUE);
}


ChipArrayClass::StripBadProximityValues(int H)
{
        int i,j;
    long U;
    int c;
    BlockClass *TempBlock;

    GlobalHeight = H;

        for (i=0; i<Xdim; i++)
    {
                for (j=0; j<Ydim; j++)
        {
                if (ValidBlock(i,j,1, H))  // note that blanks could mess this up badly!
            {
               U = DataGrid[i][j]->cdldata.unit;
                       c = CountUnitInArea(U,i-Radius, j-Radius, 2*Radius+1, 2*Radius+1);
                 if (c>MaxAllowed)
                       StripValidBlock(i,j,H);
            }
        }
    }
    // now we've got all our trouble removed from the chip
    printf("Bad Proximity values: %d %d %d %ld\n", H, Radius, MaxAllowed,
ValidBlockStack.BlockCurSize);
    return(TRUE);
}



ChipArrayClass::DoubleReplacement(long searchlimit)
{
        // idea is to "dilute" any bad values
    // this only works if the chip is sufficiently large
    // and there are sufficiently few bad items
        StripBadProximityValues(GlobalHeight);
        while (ValidBlockStack.BlockCurSize>0)
    {
         StripRandomBlocks(ValidBlockStack.BlockCurSize+100, GlobalHeight); // get some good random
locations freed up
        Shuffle(); // rearrange life
        DiagonalReplacement(searchlimit); // put 'em back, - if too much search, goes back exactly
to bad spots
                StripBadProximityValues(GlobalHeight); // find out if we've got them all
    }
}



ChipArrayClass::DiagonalReplacement(long searchlimit)
{
        // replaces blocks from stack onto the chip
    int X, Y;
    long Best;
    double EdgesAdded = 0.1;
    double TotalAdded = 0.1;
    double AvgEdges =0.1;
    double WorstEdges=0.1;
    double bestc;
    double avg;
```

```
    double worstc;
    long Report = 1000000;

    Report /=searchlimit;
    if (Report>1000)
       Report=1000;

    X=Y=0;
    while (FindNextDiagonalSlot(X,Y))
    {
       // found a location where a block was removed
        SearchLocationWithStats(X,Y,searchlimit, Best, bestc, avg, worstc);
        // found the best thing to put there

        // and so put it there!
        PlaceBlockFromStack(Best,X,Y);

        EdgesAdded += bestc; AvgEdges += avg; WorstEdges += worstc; TotalAdded ++;

        if (ValidBlockStack.BlockCurSize%Report==0)
              printf("At: %d %d %ld %lf %lf %lf\r", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    }
    printf("\nAt: %d %d %ld %lf %lf %lf\n", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    return(TRUE);
}

ChipArrayClass::AggregateReplacement(long searchlimit)
{
       // replaces blocks from stack onto the chip
    int X, Y;
    long Best;
    double EdgesAdded = 0.1;
    double TotalAdded = 0.1;
    double AvgEdges =0.1;
    double WorstEdges=0.1;
    double bestc;
    double avg;
    double worstc;
    long Report = 1000000;

    Report /=searchlimit;
    if (Report>1000)
       Report=1000;

    X=Y=0;
    while (FindNextAggregateSlot(X,Y) && (ValidBlockStack.BlockCurSize>0))
    {
       // found a location where a block was removed
        SearchLocationWithStats(X,Y,searchlimit, Best, bestc, avg, worstc);
        // found the best thing to put there

        // and so put it there!
        PlaceBlockFromStack(Best,X,Y);

        EdgesAdded += bestc; AvgEdges += avg; WorstEdges += worstc; TotalAdded ++;

        if (ValidBlockStack.BlockCurSize%Report==0)
              printf("At: %d %d %ld %lf %lf %lf\r", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    }
    printf("\nAt: %d %d %ld %lf %lf %lf\n", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    return(TRUE);
}


ChipArrayClass::HorizontalReplacement(long searchlimit)
{
```

```
        // replaces blocks from stack onto the chip
    int X, Y;
    long Best;
    double EdgesAdded = 0.1;
    double TotalAdded = 0.1;
    double AvgEdges =0.1;
    double WorstEdges=0.1;
    double bestc;
    double avg;
    double worstc;
    long Report = 1000000;

    Report /=searchlimit;
    if (Report>1000)
        Report=1000;

    X=Y=0;
    while (FindNextHorizontalSlot(X,Y))
    {
        // found a location where a block was removed
        SearchLocationWithStats(X,Y,searchlimit, Best, bestc, avg, worstc);
        // found the best thing to put there

        // and so put it there!
        PlaceBlockFromStack(Best,X,Y);

        EdgesAdded += bestc; AvgEdges += avg; WorstEdges += worstc; TotalAdded++;

        if (ValidBlockStack.BlockCurSize%Report==0)
              printf("At: %d %d %ld %lf %lf %lf\r", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    }
    printf("\nAt: %d %d %ld %lf %lf %lf\n", X, Y, ValidBlockStack.BlockCurSize,
EdgesAdded/(2*TotalAdded), EdgesAdded/AvgEdges,EdgesAdded/WorstEdges);
    return(TRUE);
}

ChipArrayClass::StripValidBlock(int X, int Y, int H)
{
       BlockClass *TempBlock;

    if (ValidBlock(X,Y, 1, H))
    {
        TempBlock = new BlockClass;
        StripAreaToBlock(TempBlock, X,Y,1,H); // take probe from chip
        TempBlock = ValidBlockStack.PutBlockOnStack(TempBlock);
        if (TempBlock!=NULL)
        {
              printf("Failure to put on stack! %d %d\n", X,Y);
        }
    }
}

ChipArrayClass::StripAllValidBlocks(int H)
{
       int i,j;
    BlockClass *TempBlock;

       for (i=0; i<Xdim; i++)
    {
        for (j=0; j<Ydim; j++)
        {
              StripValidBlock(i,j,H);
        }
        printf("Stripped: %ld\r", ValidBlockStack.BlockCurSize);
    }
}

ChipArrayClass::StripRandomBlocks(long Num, int H)
{
       long i;
```

```
    int X,Y;

        for (i=0; i<Num; i++)
    {
        if(PickRandomValidBlock(X,Y,1,H))
                StripValidBlock(X,Y,H);
    }
    return(TRUE);
}


ChipArrayClass::StripAreaToBlock(BlockClass *TempBlock, int X, int Y, int Width, int Height)
{
        if (X+Width>Xdim || Y+Height>Ydim || X<0 || Y<0)
        return(FALSE);
        // strip an area of the chip into a block
    TempBlock->Allocate(Width*Height);
    TempBlock->WSize = Width;
    TempBlock->HSize = Height;

    int counter = 0;

    int i,j;
    for (i=0; i<Width; i++)
    for (j=0; j<Height; j++)
    {
        TempBlock->DataStack[counter] = DataGrid[X+i][Y+j];
        DataGrid[X+i][Y+j] = NULL; // removed
        TempBlock->DataStack[counter]->SetRelative(i,j);
        counter++;
    }
}

ChipArrayClass::CheckBlockFitToArea(BlockClass *TempBlock, int X, int Y)
{
        int valid = TRUE;
    int i;
    int tx, ty;

    if (TempBlock==NULL)
        return(FALSE);

        for (i=0; i<TempBlock->DataStackSize && valid; i++)
    {
        if (TempBlock->DataStack[i]!=NULL)
        {
                tx = X+TempBlock->DataStack[i]->relx;
                ty = Y+TempBlock->DataStack[i]->rely;
            if (tx<Xdim && ty<Ydim && tx>-1 && ty>-1)
                                if (DataGrid[tx][ty]!=NULL)
                        valid = FALSE;
                 else
                        valid = TRUE;
            else
                valid = FALSE;
        }
    }
    return(valid);
}

ChipArrayClass::PutBlockInArea(BlockClass *TempBlock, int X, int Y)
{
        int i;
    int tx, ty;
        if (!CheckBlockFitToArea(TempBlock, X, Y))
        return(FALSE);
        for (i=0; i<TempBlock->DataStackSize; i++)
    {
        if (TempBlock->DataStack[i]!=NULL)
        {
                tx = X+TempBlock->DataStack[i]->relx;
```

```
                ty = Y+TempBlock->DataStack[i]->rely;
                        DataGrid[tx][ty] = TempBlock->DataStack[i];
                TempBlock->DataStack[i]=NULL;
            }
        }
            TempBlock->DeAllocate(); // toast!
            return(TRUE);
}

ChipArrayClass::ChipArrayClass()
{
            DataGrid = NULL;
        Xdim = Ydim = SynthSteps = 0;
        Radius = 9;
            MaxAllowed = 4;
        GlobalHeight = 2;

        ScanRadius = 1;
        LeakageHalfLife = 1;
        weightflag = 0;
}

ChipArrayClass::~ChipArrayClass()
{
            DeAllocate();
}

ChipArrayClass::Allocate(int X, int Y)
{
            DataGrid = new LocalDataClass ** [X];
        if (DataGrid==NULL)
            return(FALSE);
        int i,j;

        for (i=0; i<X; i++)
        {
            DataGrid[i] = new LocalDataClass * [Y];
            if (DataGrid[i]==NULL)
                    return(FALSE);
            for (j=0; j<Y; j++)
            {
                    DataGrid[i][j] = new LocalDataClass; // featherweight objects
                if (DataGrid[i][j]==NULL)
                    return(FALSE);
            }
        }
        Xdim = X;
        Ydim = Y;
        return(TRUE);
}

ChipArrayClass::DeAllocate()
{
            if (DataGrid==NULL)
            return(TRUE);
            int i,j;
            for (i=0; i<Xdim; i++)
        {
            for (j=0; j<Ydim && DataGrid[i]!=NULL; j++)
            {
                    if (DataGrid[i][j]!=NULL)
                            delete DataGrid[i][j];
            }
            if (DataGrid[i]!=NULL)
                    delete[] DataGrid[i];
        }
        delete[] DataGrid;
        DataGrid = NULL;
        Xdim = Ydim = 0;
        SynthSteps = 0;
}
```

```
ChipArrayClass::ReadCdl(char *FileName, int realflag)
{
        FILE *ifp;
    int maxX, maxY, X, Y;
    char datastring[MXLINE];
    int flag=TRUE;

    if (realflag)
         flag = ReadCdl(FileName, FALSE);
    if (!flag)
        return(FALSE);
        ifp = fopen(FileName, "rt");
    if (NULL==ifp)
    {
        printf("Unable to open: %s\n", FileName);
        exit(1);
    }
    fgets(datastring, MXLINE, ifp);
    maxX = 0;
    maxY = 0;
    while (!feof(ifp) && !ferror(ifp))
    {
        fgets(datastring, MXLINE, ifp);
        if (feof(ifp) || ferror(ifp))
               break;
        sscanf(datastring, "%d %d", &X, &Y);
        if (X>maxX)
               maxX=X;
        if (Y>maxY)
               maxY=Y;
        if (realflag)
               DataGrid[X][Y]->cdldata.LineScan(datastring);
        if (X==0)
               printf("%d\r", Y);
    }
    fclose(ifp);
    if(!realflag)
    {
           maxX++;
           maxY++;
        flag = Allocate(maxX, maxY);
        return(flag);
    }
    return(TRUE);
}

ChipArrayClass::DumpCdl(char *FileName)
{
        FILE *fp;
    int i,j;

        fp = fopen(FileName, "wt");
    fprintf(fp,
"X\tY\tPROBE\tDESTYPE\tFEATURE\tQUALIFIER\tEXPOS\tTBASE\tENDPOS\tPOSITION\tPBASE\tFINISHPOS\tFIXED\t
VARIABLE\tUNIT\tBLOCK\tATOM\tREPEAT\tSEQNO\tLAYOUT\tACCESSION\tLOCUS\n");
        for (j=0; j<Ydim; j++)
    {
               for (i=0; i<Xdim; i++)
        {
               if (DataGrid[i][j]!=NULL)
                      DataGrid[i][j]->cdldata.DumpLine(fp, i,j);
            else
            {
               printf("Null value in grid %d %d\n", i,j);
            }
        }
        printf("OutCdl: %d\r", j);
    }
        fclose(fp);
```

```
}

ChipArrayClass::GenerateMutFile(char *FileName)
{
        FILE *fp;
    int i,j;

        fp = fopen(FileName, "wt");
        for (j=0; j<Ydim; j++)
    {
                for (i=0; i<Xdim; i++)
        {
                if (DataGrid[i][j]!=NULL)
            {
                            DataGrid[i][j]->cdldata.DumpMut(fp); // single descriptive character
            }
            else
            {
               fprintf(fp, "-");
               printf("Null value in grid %d %d\n", i,j);
            }
        }
        fprintf(fp, "\n");
        printf("MUT: %d\r", j);
    }
        fclose(fp);
}

ChipArrayClass::GenerateDiffFile(char *FileName)
{
        FILE *fp;
    int i,j;
    int tn,te,ts,tw;
    double n,e,s,w;
    double count;

    fp = fopen(FileName, "wt");
    for (j=0; j<Ydim; j++)
    {
        for (i=0; i<Xdim; i++)
        {
                if (Valid(i,j))
            {
               fprintf(fp, "X:%d\tY:%d\t", i,j);
                tn=ts=tw=te =0;
                if (Valid(i,j-1))
                      tn = DataGrid[i][j-1]->retdata.Diff(DataGrid[i][j]->retdata);
                if (Valid(i,j+1))
                      ts = DataGrid[i][j+1]->retdata.Diff(DataGrid[i][j]->retdata);
                if (Valid(i-1,j))
                      tw = DataGrid[i-1][j]->retdata.Diff(DataGrid[i][j]->retdata);
                if (Valid(i+1,j))
                      te = DataGrid[i+1][j]->retdata.Diff(DataGrid[i][j]->retdata);
                fprintf(fp, "N:%d\tE:%d\tS:%d\tW:%d\tT:%d\t",tn,te,ts,tw,tn+te+ts+tw);
                              fprintf(fp, "LAST:%d\t", DataGrid[i][j]->retdata.GetLast());
                fprintf(fp, "BREADTH:%d\t", DataGrid[i][j]->retdata.GetLast()-DataGrid[i][j]-
>retdata.GetFirst());
                DataGrid[i][j]->PrintLongSeq(fp);
                fprintf(fp, "\t%s\n", DataGrid[i][j]->cdldata.qualifier);

                count++;
                n+=tn;
                s+=ts;
                e+=te;
                w+=tw;
            }
            else
            {
               printf("Null value in grid %d %d\n", i,j);
            }
        }
```

```
        printf("Diff: %d %lf %lf %lf %lf %lf\r", j, n/count, e/count,s/count, w/count,
(n+e+s+w)/(4*count));
    }
    //fprintf(fp,"Diff: %d %lf %lf %lf %lf %lf\n", j, n/count, e/count,s/count, w/count,
(n+e+s+w)/(4*count));
    fclose(fp);
}

ChipArrayClass::ReadRet(char *FileName, int realflag)
{
        FILE *fp;
    int i,j, k;
    char dataline[MXLINE];
    long total;

    if (realflag)
        ReadRet(FileName, 0);
    fp = fopen(FileName, "rt");
    if (fp==NULL)
    {
        printf("Unable to open: %s\n", FileName);
        exit(1);
    }

    if (realflag)
    {
        for (i=0; i<Xdim; i++)
        for (j=0; j<Ydim; j++)
        {
               if (DataGrid[i][j]!=NULL)
                               DataGrid[i][j]->retdata.Allocate(SynthSteps);  // allocate this data
            else
               printf("Death by lack of allocation\n");
        }
    }


    k=-1;
    j=Ydim;
    total = 0;

    while (fgets(dataline, MXLINE, fp))
    {
        if (dataline[0]=='r')
        {
               sscanf(dataline, "reticle: %s", &retname);  // set up reticle template name
            retname[strlen(retname)-2] = '\0';
                       // initialize for reading next lines
            k++;
            printf("Reticle: %d\r", k);
            j=Ydim;
            continue;
        }
        if (strlen(dataline)<10 || dataline[0]==';')
            continue;
        if (dataline[0]=='0' || dataline[0]=='1')
        {
               j--;
               for (i=0; i<Xdim && j>-1; i++)
            {
               if (dataline[i]=='1')
                {
                       if (realflag)
                    {
                               DataGrid[i][j]->retdata.SetBit(1,k);
                    }
                    total ++;
                }
            }
        }
    }
```

```
    NumOnes = total;
    SynthSteps = k+1;

    fclose(fp);
}

ChipArrayClass::DumpRet(char *FileName)
{
       FILE *fp;
    int i,j,k;
    char dataline[MXLINE];

       fp = fopen(FileName, "wt");

    fprintf(fp, "; This file has been annealed to minimize edges\n");

    for (k=0; k<SynthSteps; k++)
    {
              fprintf(fp,"\n\n;base: X");
              fprintf(fp,"\nreticle: %s%02d", retname, (k+1));
              fprintf(fp,"\nR I 1 1 0 0 %d %d %d",Xdim, Ydim, 1);
       for (j=Ydim-1; j>-1; j--)
       {
           fprintf(fp, "\n");
              for (i=0; i<Xdim; i++)
           {
              if (DataGrid[i][j]==NULL)
               {
                     printf("Data leakage: %d %d\n", i,j);
               }
               else
                     if (DataGrid[i][j]->retdata.GetBit(k))
                            dataline[i] = '1';
                      else
                            dataline[i] = '0';
           }
           dataline[Xdim] = '\0';
           fprintf(fp, "%s", dataline);
       }
       fprintf(fp,";\n0;\n");
       printf("DUMPRET: %d\r", k);
    }
    fclose(fp);
}

ChipArrayClass::InterpretInstructionLine(char *Line)
{
       char TempStr[MXLINE];
    int height;
    long searchlimit;
    long start, finish;
    int tx,ty,x,y;
    int value;
    int destype;
    int radius, max;
    double dval;

       // read an instruction and do the appropriate thing
    if (Line[0]==';')
       return(TRUE); // comment
    sscanf(Line, "%s", TempStr); // pick off the initial piece
    if (!strcmp(TempStr, "READCDL:"))
    {
              sscanf(Line, "READCDL: %s", TempStr);
        ReadCdl(TempStr, TRUE);
        return(TRUE);
    }
    if (!strcmp(TempStr, "READRET:"))
    {
       sscanf(Line, "READRET: %s", TempStr);
        ReadRet(TempStr,1);
```

```
    return(TRUE);
}
if (!strcmp(TempStr, "DUMPCDL:"))
{
    sscanf(Line, "DUMPCDL: %s", TempStr);
    DumpCdl(TempStr);
    return(TRUE);
}
if (!strcmp(TempStr, "DUMPRET:"))
{
    sscanf(Line, "DUMPRET: %s", TempStr);
    DumpRet(TempStr);
    return(TRUE);
}
if (!strcmp(TempStr, "DUMPMUT:"))
{
    sscanf(Line, "DUMPMUT: %s", TempStr);
    GenerateMutFile(TempStr);
    return(TRUE);
}
if (!strcmp(TempStr, "DUMPDIFF:"))
{
    sscanf(Line, "DUMPDIFF: %s", TempStr);
    GenerateDiffFile(TempStr);
    return(TRUE);
}
    if (!strcmp(TempStr, "STRIPBLOCKS:"))
{
    sscanf(Line, "STRIPBLOCKS: %d", &height);
    GlobalHeight = height;
    StripAllValidBlocks(height);
    Shuffle();
    return(TRUE);
}
if (!strcmp(TempStr, "DIAGONALREPLACEMENT:"))
{
    sscanf(Line, "DIAGONALREPLACEMENT: %ld", &searchlimit);
    DiagonalReplacement(searchlimit); // put all blocks back on chip
    return(TRUE);
}
if (!strcmp(TempStr, "HORIZONTALREPLACEMENT:"))
{
    sscanf(Line, "HORIZONTALREPLACEMENT: %ld", &searchlimit);
    DiagonalReplacement(searchlimit); // put all blocks back on chip
    return(TRUE);
}
if (!strcmp(TempStr, "AGGREPLACEMENT:"))
{
    sscanf(Line, "AGGREPLACEMENT: %ld", &searchlimit);
    AggregateReplacement(searchlimit); // put all blocks back on chip
    return(TRUE);
}
if (!strcmp(TempStr, "SETVALIDUNITS:"))
{
    sscanf(Line, "SETVALIDUNITS: %ld %ld %d", &start, &finish, &value);
    SetUnits(start, finish, value);
    return(TRUE);
}
if (!strcmp(TempStr, "SETVALIDAREA:"))
{
    sscanf(Line, "SETVALIDAREA: %d %d %d %d %d", &x,&y,&tx,&ty, &value);
    SetArea(x,y,tx,ty,value);
    return(TRUE);
}
if (!strcmp(TempStr, "SETVALIDANTIAREA:"))
{
    sscanf(Line, "SETVALIDANTIAREA: %d %d %d %d %d", &x,&y,&tx,&ty, &value);
    SetAntiArea(x,y,tx,ty,value);
    return(TRUE);
}
if (!strcmp(TempStr, "SETVALIDDESTYPE:"))
```

```
    {
        sscanf(Line, "SETVALIDDESTYPE: %ld %d", &destype, &value);
        SetDestype(destype, value);
        return(TRUE);
    }
    if (!strcmp(TempStr, "STRIPBADPROXIMITY:"))
    {
        sscanf(Line, "STRIPBADPROXIMITY: %d %d %d", &height, &radius, &max);
        Radius = radius;
        MaxAllowed = max;
        StripBadProximityValues(height);
        return(TRUE);
    }
    if (!strcmp(TempStr, "SETPROXIMITY:"))
    {
        sscanf(Line, "SETPROXIMITY: %d %d", &radius, &max);
        Radius = radius;
        MaxAllowed = max;
        return(TRUE);
    }
    if (!strcmp(TempStr, "FIXBAD:"))
    {
        sscanf(Line, "FIXBAD: %ld", &searchlimit);
        DoubleReplacement(searchlimit);
        return(TRUE);
    }
    if (!strcmp(TempStr, "WEIGHT:"))
    {
        sscanf(Line, "WEIGHT: %d %lf", &radius, &dval);
        ScanRadius = radius;
        LeakageHalfLife = dval;
        weightflag = TRUE; // use weights
        return(TRUE);
    }
    if (!strcmp(TempStr, "NOWEIGHT:"))
    {
        weightflag = FALSE;
        return(TRUE);
    }
    return(FALSE);
}

ChipArrayClass::ReadInstructionFile(char *FileName)
{
        // read the file and be happy
    FILE *fp;
    char dataline[MXLINE];

    fp = fopen(FileName, "rt");
    if (fp==NULL)
        return(FALSE);
    while (fgets(dataline, MXLINE, fp))
    {
            InterpretInstructionLine(dataline);
    }
    fclose(fp);
}


TestTwo()
{
        ChipArrayClass Test;
    Test.ReadInstructionFile("test.opt");
}

Live(char *FileName)
{
        ChipArrayClass Test;
    Test.ReadInstructionFile(FileName);
}
```

```
main(int argc, char **argv)
{
    if (argc==2)
    {
        Live(argv[1]);
    }
    else
    {
        printf("File name required!");
    }
};
```